

Type-Safe Covariance in C++

VITALY SURAZHSKY JOSEPH (YOSSI) GIL*
Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

vitus | yogi @cs.technion.ac.il

September 29, 2004

Abstract

We present a programming technique for implementing type safe covariance in C++. In a sense, we implement most of Bruce’s *matching* approach to the covariance dilemma in C++. The appeal in our approach is that it relies on existing mechanisms, specifically templates, and does not require any modification to the existing language. The practical value of the technique was demonstrated in its successful incorporation in a large software body. We identify the ingredients of a programming language required for applying the technique, and discuss extensions to other languages.

1 Introduction

Two clashing forces make the recalcitrant covariance dilemma. On the one hand, virtually all modeling situations are *covariant* in nature, i.e., a specialization in one hierarchy is correlated with a specialization in another hierarchy. A problem-world example is the **child** specialization of a **person**, which is correlated with a specialization of **physician** to a **pediatrician**. A program-world example is the specialization of **singly-linked-list** into **doubly-linked-list**, correlated with the specialization of **singly-linked-node** into **doubly-linked-node**. Famous are also cases of *auto-covariance* in which changes a hierarchy is correlated with itself. For example, comparing instances of **point** for equality is specialized into comparing instances of **color-point**.

On the other hand, it is impossible to statically type-check inclusion polymorphism [7] when covariance is allowed. Suppose for example that instances of **child** can be freely used at any place where instances of **person** might. Then, one of these places, might associate a certain **physician** who is not a **pediatrician** with a **child**. Such an association can only be prevented with run-time type checks. (For a more detailed exposition of covariance see sections 3–4 in Kim Bruce’s lecture notes ¹ on the topic.)

In dealing with this dilemma, different programming languages in practical use take different approaches. Favoring modeling convenience language such SMALLTALK [14] and CLOS [24] do not impose static type checking. Favoring static typing over modeling convenience some languages, such as early versions of C++ [25], forbid covariance altogether. Other languages allow it a restricted form. For example, current C++ [26] permits covariant changes only to function return type, since this can be statically checked. SATHER [27, 21] extends this by allowing *contravariant* changes to arguments, which is not very useful for modeling, but can be statically checked.

A combination of static and dynamic checks can also be found. In the JAVA [1] case, even though covariance cannot be declared or statically checked, it is possible for a programmer to use a downcast relying on a covariance presumption. The runtime system will then detect all type errors resulting from making this assumption. EIFFEL [17] is unique in allowing covariant definitions, using what is called *anchored types*. Type safety is then achieved by a

*Contact author

¹<http://www.cs.williams.edu/~kim/README.html#Static>

link-time dataflow analysis, known as “system validity check”. Apparently, system validity check was never part of commercial EIFFEL compilers, which therefore compromise static type safety.

Virtual types in BETA [16] are yet another form of covariance. Even though BETA is in general type safe, the type correctness of those aspects of the language which touch virtual types are ensured by runtime checks.

In summary, none of these languages succeeds in combining type safety and the convenience of covariant modeling.

Stepping beyond current languages, type theorists dedicated much attention to the problem. The exact types approach [2] restricts covariance as follows. In each call of the form

$$c.m(d) \tag{1}$$

it is required that if m is covariantly overridden, either c must be of an *exact type* C_1 , i.e., it is not allowed to be of any type $C_2 \prec C_1$, $C_2 \neq C_1$. Meyer’s polymorphic catcalls² are a variation in which dataflow analysis replaces an exact type declaration in guaranteeing that c is monomorphic. An experimental implementation of exact types was in EIFFEL recently reported [10].

Catagana [8] on the other hand convincingly argued for *encapsulated multi-methods* in which the polymorphic nature of (1) is enriched rather than restricted. Consider a definition of a method m in class C_1 ,

$$m(a : D_1) = \dots \tag{2}$$

with a covariant definition in class $C_2 \prec C_1$

$$m(a : D_2) = \dots \tag{3}$$

where $D_2 \prec D_1$. Then, definition (3) *adds* to (2), rather than merely overriding it. The polymorphic call (1) is implemented as a *multi-dispatch*: if $c \in C_2$ and $d \in D_2$, then the implementation (3) is invoked. Otherwise, the implementation (2) is used. Multi-methods support subtyping, static typing and covariance, at the price of deviating from the single-dispatch tradition of object oriented languages. Also, multi-methods do not support covariant changes to field types.

Problems such as type-checking [9], and especially in a separate compilation environment [19], compounded by the non-OO semantics, are most likely the reason that multi-methods did not yet find their way into mainstream programming languages. The VISITOR pattern [12] in its many incarnations is nothing but a programming technique for implementing multi-methods in single-dispatch languages.

Matching, the other major type-theoretical approach, resolves the covariance dilemma by slightly weakening the notion of subtyping and somewhat restricting runtime substitutability. Matching was demonstrated in research languages such as **PolyTOIL** [6], *LOOM* [5] and LGM [23]. Matching is a weaker notion than the subtyping in the sense that if a type matches another, then it cannot be substituted freely for it. Covariant specialization is on the other hand allowed along the matching relationship. With matching, inclusion polymorphism gives way to *match bounded polymorphism* which dwells on the notion of *match type*, the type of all entities of of types matching a certain type.

The contribution in this paper is a programming technique for implementing matching statically typed contemporary languages, specifically C++, without any modification to the language syntax and semantics. Our type safe implementation is used successfully in a large application for managing geometrical and graph theoretical entities.

The techniques uses familiar programming techniques, specifically template programming similar, but of a smaller scale than STL [20] or compile time symbolic derivation [13]. We therefore believe that the technique would be appealing to some programmers than the more theoretical advances. After learning the basic notions, the programmer essentially in C++.

Also worthy of notice is that the technique combines the flexibility advantage of dynamic typing with the safety of static typing. The flexibility and the convenience of using the technique comes from the fact that the programmer does not have to define or understand sophisticated type declarations, drawn from a rich and powerful type system. However, since the technique used templates and template expansion, the body of computation is carried out at compile time. “type errors” will result in the production of compiler error messages, and in failure to generate the

²<http://www.eiffel.com/doc/manuals/technology/typing/cat.html>

covariant classes. Hence, the programmer is saved the trouble of making a type correct mutual covariant definition of a matching hierarchy base, which should guarantee that all descendants of a certain type are type correct. Instead, the programmer tries to define each class in the hierarchy in its turn. It is not guaranteed that each class defined will be type correct. Each such class will ingratiate some templates. “Run time” type errors, as a result our lack to ability the sophisticated type system, will make “run time” errors. However, since template expansion is at compile time, these errors will be emitted by the compiler, as compiler errors.

Outline Sec. 2 uses the example of a covariant graph hierarchy to introduce the concept of configuration classes, and the way they are used in the implementation. To make a smooth discourse, the presentation uses at this stage several C++ macros, designed to hide some of the C++ lingo. We then continue in Sec. 3 to discuss auto covariance, and in comparing our technique with *LOOM*. In Sec. 4 reviews the two examples from a more programming language theoretical perspective. In particular, this section discusses the notions of recursive and collateral definitions, and the way their realization in C++ makes our solution possible. Sec. 4 also reveals the C++ macros used in Sec. 2. In the final Sec. 5 we discuss the results, trying to explain concludes the paper with some directions for future research.

2 A Covariant Graph Hierarchy Example

In this section, we give a quick introduction to our solution of the covariance dilemma, showing how it might be used by a C++ programmer. For the purpose of exposition some of the intricate details are hidden behind `#define` macros. The fine details will be revealed in Sec. 4.

Most examples in the literature revolve around the theme of auto-covariance, such as a class `Point` with an `equal` method, specialized by `ColorPoint`. We will start with the more general case, as illustrated in Fig. 2.1.

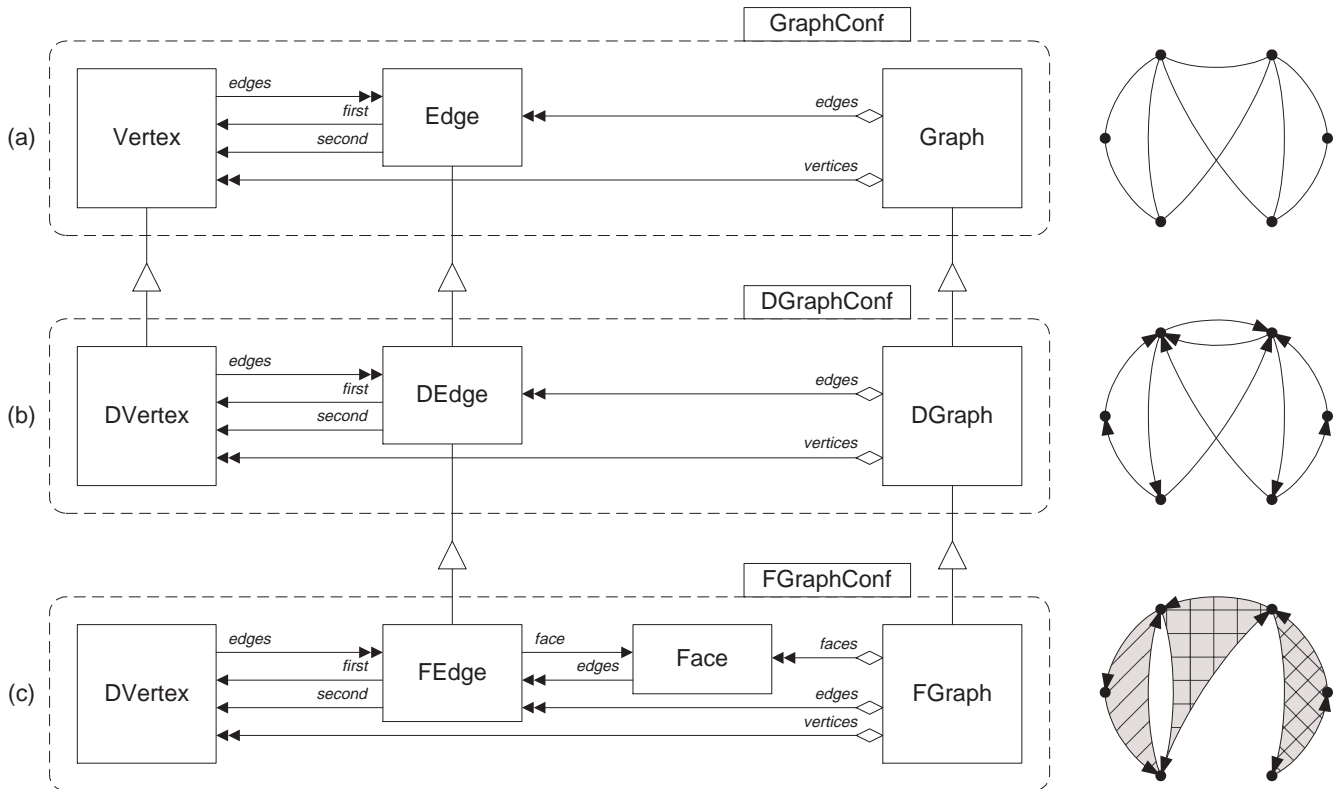


Figure 2.1: The Graphs diagrams

The figure illustrates a covariant hierarchy modeling various entities in graph theory. The hierarchy is drawn from

a 10KLOC C++ library of algorithms of 2D shape morphing in computer graphics [15].³ The two classes `Vertex` and `Edge` comprise a graph from the graph theory and describe the basic behavior and functionality.

At the top of the figure (a) we see that a `Graph` has a set of instances of `Edge` and a set of instances `Vertex`. Each `Edge` has two references (`first` and `second`) to vertices. A `Vertex` has a list of references to the edges incident on it. At the middle section (b) of the figure, we see that `DGraph` modeling a directed graph, specializes `Graph`. A directed graph has directed edges, and directed vertices. Thus, `DVertex` and `DEdge` specialize `Vertex` and `DEdge` respectively.

The bottom section (c) of the figure models the concept of *face graph*, which is a directed planar graphs augmented with face information. A *face* is a cyclic sequence of edges forming a simple cycle in a graph. Intuitively, an embedding of a planar graph in the plane subdivides the plane into regions. Each region is represented by a face, which also defines an order for traversing the edges surrounding it. The specialization of `DGraph` in `FGraph` is correlated with a specialization of `DEdge` into `FEdge`. Note that `FEdge` stores a reference to `Face`—a new concept which does not exists in directed graphs.

Fig. 2.2 shows the definitions for the face graph. Class `FGraph` being derived from `DGraph` holds objects of vertices, edges and faces, and extends the functionality of class `DGraph`. The vertices of the face graph has the same behavior and structure as those of the directed graph except for the type of its adjacent edges. The configuration definition of the face graph reflects the usage of class `DVertex` in the arrangement of the face graph classes. Instances of type `FGraph` and of type `Face` are created to demonstrate the usage of the definitions.

The C++ implementation of Fig. 2.1 is presented in figures 2.2–2.7.

In implementing covariance, the programmer must be thinking in terms of actors, roles, and configurations. The three concepts are not supported as such in C++, hence the use of macros, which are all written in all capitals. An actor may interact with roles, and assume other roles. Actors may be specialized. Covariance specialization of an actor is realized by changing the actors which assume the roles with which this actor interacts. (Later we will see that actors nothing but are generic classes, or class templates as they are known in the C++ jargon.)

Fig. 2.2 illustrates the definition of actors.

```

ACTOR Edge { public:
    INTERACTS_WITH(V);
    Edge(V& v1, V& v2);           // a covariant constructor
    void set_first(V* v);         // a covariant method
    void set_second(V* v);        // another covariant method
    V& get_first();               // a method with a covariant return value
    // more definitions ...
    protected: V *first, *second; // two covariant fields
};
ACTOR Vertex { public:
    INTERACTS_WITH(E);
    void insert_edge(E *e);       // a covariant method
    void delete_edge(E *e);       // another covariant method
    // ...
    protected: list<E *> edges;   // a covariant field
};
ACTOR Graph { public:
    INTERACTS_WITH(V);
    INTERACTS_WITH(E);
    Graph(list<V *>& vs, list<E *>& es); // a doubly covariant constructor
    // ...
    protected: list<V *> vertices; list<E *> edges;
};

```

Figure 2.2: The actors of an undirected graph

Three actors: `Edge`, `Vertex` and `Graph` are defined in the figure. If the type of an field, a method argument, or

³We hope that this real-life example is more appealing to our programming intuition than Animal-Hebivore-Dish-Food sort of examples sometimes used in the literature. There is no fundamental between the two.

function return value in an actor may take covariant specialization, then this type must be declared as a *role*. (Later we will see that roles are realized as **typedefs** in a class passed as an argument to the class template which realizes an actor.) A definition of an actor starts with a series of `INTERACTS_WITH` directives, each declaring a role with which the actor may interact. There are three roles with which the actors in Fig. 2.2 interact: `G` designating graphs, `V` designating vertices, and `E` designating edges. Actor `Graph`, for example, interacts with roles `V` and `E`.

Other than stating that a role is a type, the role declaration does not place any explicit constraints on it. Once the interacting roles are defined, the definition of an actor is similar to a C++ class definition, where roles can be used anywhere types are used. For example, actor `Edge` has a covariant constructor taking two arguments whose type is the role `E`, methods `set_first` and `set_second` each taking a covariant argument, a method `get_first` with a covariant return type, two covariant field definition (`first` and `next`), and possibly more covariant and nonvariant member definitions.

The body of an actor may, and usually will, place implicit constraints on the roles it interacts with. In the figure, actor `Edge` makes the assumption that type `V` is such that both `V *` and `V &` are valid types. This assumption does not hold for example for the type `int &`. More interesting assumptions on `V` are made by, say, the body of the method `set_first(V)`:

```
set_first(V* v)
{
    if (first != 0)
        first->delete_edge(this);
    first = v;
    v->insert_edge(this)
}
```

It is assumed in the above code excerpt that role `V` has two methods named `delete_edge` and `insert_edge` which can receive a parameter of the type of **this**. For a set of actors A which interact with a set of roles R , let $c(A)$ be the set of assumptions that A makes on R .⁴

Actors are made into concrete classes, and used to instantiate objects, only after roles are assigned to them. A *configuration* is a simultaneous assignment of roles to actors, creating a set of cooperating classes. (Later we will see that configurations are realized as a series of **typedefs** made in a class passed as a template parameter to an actor.)

Mathematically, a configuration is a mapping the set R of roles onto the set A of actors, such that each role is assigned to exactly one actor. An actor may however assume more than one role. The assignment must be consistent in the sense that the set of constraints $c(A)$ must hold for the set of role-actor pairs.

Consider for example the configuration of undirected graphs, as illustrated in Fig. 2.3.

```
CONFIGURATION(GraphConfig)
    ASSIGN_ROLE(V, Vertex)
    ASSIGN_ROLE(E, Edge)
    ASSIGN_ROLE(G, Graph)
END_CONFIGURATION
```

Figure 2.3: A configuration of an undirected graph

Configuration `GraphConfig` assigns the roles `V` (a vertex) `E` (an edge) and `G` (a graph) respectively to actors `Vertex`, `Edge` and `Graph`. The above role assignment does a *simultaneous* substitution in Fig. 2.2, to create three ordinary C++ class, out of which objects can be instantiated as in Fig. 2.4.

```
GraphConfig::V v1, v2, v3;
GraphConfig::E e12(v1,v2), e23(v2, v3);
GraphConfig::G g(CONS(&v1,CONS(&v2, &v3)), CONS(&e12, &e23));
```

Figure 2.4: Instantiating the `GraphConfig` configuration

⁴The definition of set $c(A)$ is deliberately very loose. We do not state what assumptions might be there, how they are structured, etc.

As we can see in the figure, selecting a certain role out of a configuration is done simply by using the `::` operator. Trying to elicit roles from a configuration in which they are not defined, will result in a compiler error. Similarly, a configuration which does not create a complete or consistent role assignment, will result in a compiler error. In general, these errors may not be easy to understand.

In the figure, we construct three directed vertices, `v1`, `v2`, and `v3`, and edges `e12` (connecting `v1` to `v2`) and `e23` (connecting `v2` to `v3`). The directed graph `g` is then constructed with the two lists `(v1, v2)` and `(e1, e2, e3)`. A LISP-like `CONS` function is used for creating the lists. The implementation of `CONS` is standard using an overloaded template function, and will not be repeated here.

Fig. 2.4 helps also understand the circular nature of the substitution in a configuration. The C++ class `GraphConfig::V` is in fact the actor `Vertex`, with the role `E` substituted by the C++ class `GraphConfig::E`. The C++ class `GraphConfig::E` is in its turn obtained by substituting the role `V` by the C++ class `GraphConfig::V` in actor `Edge`. The C++ class `GraphConfig::G` is obtained by making both these role substitutions in actor `Graph`. The definition of these three C++ classes is mutually recursive.

The configuration `GraphConfig` is consistent since actor `Edge` indeed has two functions named `delete_edge` and `insert_edge`. The assumption about the type of the arguments is satisfied by the class `GraphConfig::V`, in which the arguments to `delete_edge` and `insert_edge` are of type `GraphConfig::E`.

Circular definitions exist as such in C++. The main purpose of the distinction between actors and roles is that new actors can be specialized from existing actors, inheriting the roles they interact with. An example of actor specialization is given in Fig. 2.5, where actor `DEdge` specializes `Edge` etc.

```
SPECIALIZED_ACTOR(DEdge, Edge) { public:
    // role V is inherited
    V* source() const; V* target() const;
    // more functionality
    protected: // new internal staff
};
SPECIALIZED_ACTOR(DVertex, Vertex) { public:
    // role E is inherited
    // extend functionality ...
    protected: // new internal staff
};
SPECIALIZED_ACTOR(DGraph, Graph) { public:
    // roles V and E are inherited
    // new functionality and algorithms
    protected: // new internal staff
};
```

Figure 2.5: The actors in a directed graph

The macro `SPECIALIZED_ACTOR` creates a new actor specializing an existing one. Actor specialization is just like derivation in C++, and the specializing actor can override the definitions in the specialized actor, adding to them, etc. However, since actors are not classes, a derived actor cannot be used where the base actor is used. Hence the covariance dilemma does not appear.

Since no constraints are placed on roles, it is not necessary to change the definition the role in a derived class. The triple of covariant classes realizing the directed graph concept is realized by a configuration assigning respective roles the three actors, as done by Fig. 2.6. Again, the configuration `DGraphConfig` in this figure does a simultaneous substitution, this time on the actors defined in Fig. 2.5. Three new C++ classes `DGraphConfig::E`, `DGraphConfig::V`, and `DGraphConfig::G`, are generated from the three actors, by a consistent substituting of the roles with the newly generated classes.

Again, the definition of these three new classes is mutually recursive. The *structure* of this mutual recursion is similar to that of of the mutual recursion in the three classes generated by configuration `GraphConfig`. However, the class `DGraphConfig::V` (say) does not inherit from the class `GraphConfig::V`, even though these two classes were defined using the same role assigned to two actors bounded together by specialization.

A faced graph requires yet another actor `Face`. The complete definition of the actors in face graph is given in

```

CONFIGURATION(DGraphConfig)
  ASSIGN_ROLE(V, DVertex)
  ASSIGN_ROLE(E, DEdge)
  ASSIGN_ROLE(G, DGraph)
END_CONFIGURATION

```

Figure 2.6: A configuration of a directed graph

Fig. 2.7. Actor Face interacts with edges (role E), and stores a list of edges.

```

SPECIALIZED_ACTOR(FEdge, DEdge) { public:
  // role V is inherited
  INTERACTS_WITH(F);
  F* getFace() const { return face; }
  // more functionality
  protected: F* face;
};
ACTOR Face { public:
  INTERACTS_WITH(E);
  // ...
  protected: list<E*> edges;
};
SPECIALIZED_ACTOR(FGraph, DGraph) { public:
  INTERACTS_WITH(F); // roles V and E are inherited
  // ...
  protected: list<F*> faces;
};

```

Figure 2.7: The actors in a face graph

Note that we do not specialize the actor `DVertex`. Also note that a new role `F` is introduced in actors `FGraph` and `FEdge`. This new role makes it necessary for the configuration `FGraphConfig` in Fig. 2.8 to make four assignments of roles to actors.

```

CONFIGURATION(FGraphConfig)
  ASSIGN_ROLE(V, DVertex)
  ASSIGN_ROLE(E, FEdge)
  ASSIGN_ROLE(F, Face)
  ASSIGN_ROLE(G, FGraph)
END_CONFIGURATION

```

Figure 2.8: A configuration for a face graph

Configuration `FGraphConfig` generates four new C++ classes, defined in a mutually recursive manner. This mutual recursion is precisely the reason why the assignment of role `V` to actor `DVertex`, which occurs identically in both `DGraphConfig` and `FGraphConfig`, does not generate the same class.

3 Auto Covariance, Matching and Match Bounded Polymorphism

The running example in the previous section showed how the three ex-lingual concepts: actors, roles and configurations can be effectively used to realize a mutually covariant hierarchy with covariant changes to the types of arguments to methods, return values and fields. Even though auto-covariance can be thought of as a special case of mutual-covariance it poses some delicate points, which we explore in this section. In the course of the exploration, it will become evident that *LOOM* style matching and match-bounded polymorphism can be realized in C++ without any lingual extension.

Let us start with a variant of the familiar example of `ColorPoint` inheriting from `Point` [23].

Fig. 3.1 presents Point, the base actor in our auto covariance play.

```

ACTOR Point { public:
    INTERACTS_WITH(P);
    SELF_IS(P);

    void move(int dx, int dy) { x += dx; y += dy; }
    bool equal(const Self& other) const { return x == other.x && y == other.y;}
    Self *neighbor; // an auto covariant public data member
    protected: int x, y;
};

```

Figure 3.1: The auto-covariant actor Point

Method `equal` receives an argument of type (`const` reference to) `Self`, which will (later) be bound to the type of the `this`. The hidden (`this`) and the `other` argument to `equal` are of the same type, which explains why it is called a *binary* method, which are perhaps the purest form of the run time type unsafety of covariance [3].

The `public` data member `neighbor` is covariant since its type `Self` *.

In order to bind `Self` with the role that the actor plays, two steps are required. In the directive invocation `INTERACTS_WITH(P)` actor declares that there is a role `P` with which it interacts. All the actor knows is that in any configuration, this role may be assigned to some actor.

The directive invocation `SELF_IS(P)` uses, as a matter of convenience, a `typedef` to bind the name `Self` to the role `P`. In the actors-roles interplay, there is no way of imposing a constraint that a certain role is assigned to a certain actor. It is possible to define a directive

```
MY_ROLE(P)
```

as a short hand for `INTERACTS_WITH(P)` followed by `SELF_IS(P)`. The net result is that role `Self` is similar to Bruce et al.'s [6] `MyType`. The main difference is that the semantics of roles and actors does not impose that the actor will indeed receive its wished for role. Hence, the type name `Self` may be assigned to another actor assuming one of its roles.

Fig. 3.2 shows how an actual class, `PConfig::P`, can be created from the actor `Point`.

```

CONFIGURATION(PConfig)
    ASSIGN_ROLE(P, Point)
END_CONFIGURATION

```

Figure 3.2: A configuration of actor Point

Fig. 3.3 gives the definition of actor `ColorPoint` obtained by a specialization of actor `Point`.

```

SPECIALIZED_ACTOR(ColorPoint, Point) { public:
    SUPER_IS(Point); // define the Super:: name space
    // Role Self is inherited
    bool equal(const Self& other) const {
        return Super::equal(other) && color == other.color;}
    // public data member neighbor is inherited, with a covariant change to its type.
    protected: int color;
};

```

Figure 3.3: Specializing actor Point into actor ColorPoint

The directive (macro) `SUPER_IS` makes it possible to refer to the type of the base actor as `Super`. This directive is similar to the standard technique of defining using a `typedef` to abstract the name of the base class in C++:

```

class X: public Y {
    typedef Y inherited;
    /* ... */ }
}

```

After the invocation `SUPER_IS(Point)` it is possible to refer to members in the base actor using the `Super::` name space selector prefix. (The implementation of actors as template classes does not make it possible to refer to the base actor as `Point::`.) Note how the overriding method `equal` invokes the overridden version.

The configuration `CPConfig` in Fig. 3.4 creates the concrete class `CPConfig::P`.

```
CONFIGURATION(CPConfig)
  ASSIGN_ROLE(P, ColorPoint);
END_CONFIGURATION
```

Figure 3.4: A configuration for auto-covariant `ColorPoint`

To see how our implementation makes type-safe covariant classes consider Fig. 3.5. The figure makes two `typedef` commands to generate the more convenient names `point` and `color_point`. Class `point` has instances `p1` and `p2`, while class `color_point` has instance `cp1` and `cp2`.

Method `move` is equally applicable to instances of both classes, even though there is no subtype relationship between the two classes. Method `equal` insists on taking an argument of the type of its receiver. Failure to do so will result in a compiler generated type error.

```
typedef PConfig::P point;
typedef CPConfig::P color_point;

point p1, p2;
color_point cp1, cp2;

p1.move(2,3)           // call to Fig. 3.1 definition of equal
cp2.move(2, 3);       // call to the (actor) inherited method

color_point& rcp = p1; // type error! (color_point is not a subtype of point)

p1.equal(p2);         // call to Fig. 3.1 definition of equal
cp1.equal(cp2);       // call to Fig. 3.3 definition of equal
p1.equal(cp1);        // type error!
cp1.equal(p1);        // type error!
```

Figure 3.5: The auto-covariant classes `point` and `color_point`

What is the relationship between classes `point` and `color_point`? It is clearly not subtyping. We can say that these two classes were generated by a specializing-specialized pair of actors, when assigned the same role. In many ways `color_point` *matches* class `point`, where the term matching is used in the sense of [4]. The essential difference is a result of our strategical approach to the problem: since we add nothing to the core language, we can only hope to approximate “match types”, `MyType` and the other lingual features in *LOOM* as high level programming conventions. The challenge is in a faithful emulation of not only the features but also their consequences.

A case in point is *match-bounded polymorphism*, which was introduced in *LOOM* to make up for the lack of subtyping. Specifically, the *hash type* `# τ` is used to denote any type which matches τ . The example used in [4] is that instead of the inclusion polymorphism procedure

```
procedure setWindow(newWindow: Window)
```

accepting any argument whose type is a *subtype* of `Window`, one uses

```
procedure setWindow(newWindow: #Window)
```

accepting any argument whose type is a match for `Window`. We say that procedure `setWindow(newWindow: #Window)` uses *match-bounded polymorphism*.

Fig. 3.6 shows how this kind of polymorphism can be emulated in our system, by using the directive

```
MATCH_BOUNDED_FUNCTION
```

(which internally translates to a definition of a function template).

```
MATCH_BOUNDED_FUNCTION
void print(TYPE(Point) p) {
    // ...
}

print(p1);    // print a point
print(cp1);   // print a color_point
```

Figure 3.6: A match bounded polymorphic procedure

A question which arises naturally here is: What is the type of the argument to function `print`? The *LOOM* answer would be type `#point`, namely, the union of all types which match the concrete class `point` (Fig. 3.5). In C++ however, the class `point` does not even necessarily exist. The notation `TYPE(Point)` stresses that `print` can take an argument of any type created by a consistent assignment of a role to actor `Point` or any other actor specializing it. Function `print` is therefore an emulation of match-bounded polymorphism.

A *LOOM* variable may be of a hash-type. For example, the definition

```
var v: #Point;
```

allows storing in `v` values of type `Point`, or any other type which matches it. The restriction is of course that binary methods defined in `Point` are not part of the type `#Point`, and therefore cannot be executed on `v`.

It is only an illusion that the parameter `p` in Fig. 3.6 is of the hash type `#Point`. `print` is a template function. In each of its instantiations, there is a different such `p`. This makes it possible for function `print` to invoke binary methods on `p`, which would have not been possible if it was of a hash-type.

The reason that there are no hash-types in C++ is related to the fact actors are realized with a class template. The set of constraints made by an actor is defined only implicitly defined in the class template. In order to create a hash type in C++ the user must manually process these assumptions, selecting those which hold for all role assignments can serve as the hash type definition.

In this respect, our approach is weaker than *LOOM*. On the other hand, we offer covariant and auto covariant data members which *LOOM* lacks. An example of an auto covariant field is `neighbor` in Fig. 3.1. The following code excerpt demonstrates how this might be used.

```
p1.neighbor = &p2;
cp1.neighbor = &cp2;

cp1.neighbor = &p1;    // type error! types are point are incompatible
p1.neighbor = &cp1;    // type error! even upcasting is not permitted
```

Examples of a covariant data member are `first` and `second` in Fig. 2.2.

4 Recursive and Collateral Definitions

After having gained experience in the macro wrapped type safe covariance system, it is time to expose the inner working of the system. Since the expressive power of macros is quite limited, the exposure is essential for applying the approach in more demanding situations, such as actors with multiple inheritance.

As hinted above, our approach relies on template programming to capture the notion of an abstract relationship between classes, which must be able to suffer covariant changes. Consider for example the covariant specialization of `Person` into `Child`. Class `Person` has a field `physician` of type `Doctor`, which covariantly changes its type in a `Child` class to type `Pediatrician`. A template based approach comes then naturally, with the idea of passing the type of the `physician` as a template argument, as follows:

```

template <typename physicianType>
class PersonTemplate { public:
    physicianType physician;
    // ...
};

```

Then, the template class `PersonTemplate<Doctor>` will be the class `Person`, while

```
PersonTemplate<Pediatrixian>
```

will be the class `Child`. There are two problems in this simplistic approach. First, `Child` will be structured exactly as `Person`, without being able to override any of its methods, or add to them. We will see how this problem can be addressed by using inheritance among class templates.

The second and more serious problem with this approach is that it completely breaks down when physicians start maintaining their patients list, and pediatricians insist that only children would be allowed in their lists. The temptation is to write the the dual of the above excerpt

```

template <typename patientType>
class DoctorTemplate { public:
    list<patientType> patients;
    // ...
};

```

Again, a `Doctor` would be `DoctorTemplate<Person>` and a `Physician` a `DoctorTemplate<Child>`. A failure due to circular, unfounded, definitions occurs then even in the first `Doctor–Person` pair:

```

Person = PersonTemplate<Doctor>
        = PersonTemplate<DoctorTemplate<Person> >
        = PersonTemplate<DoctorTemplate<PersonTemplate<Doctor> > >
        = ...

```

What is lacking here is a foundation on which the mutual recursion could rely. Mutually recursive collateral declarations [28, Chap. 4] are not very common in programming languages. A notable exception is ML [22], in which a **rec** directive allows for recursive definitions, and the **and** keyword allows several definitions to occur collaterally, i.e., simultaneously. In the following

```
val rec D1 and D2
```

definition D_1 can rely on D_2 and vice versa. More common is a mechanism similar to PASCAL's [29] **forward** declaration, in which partial information on D_2 is declared first. This partial data (typically a signature) is sufficient for using D_2 in a definition D_1 .

Our vicious circle is that `Doctor` and `Person` are mutually recursive template classes; each expected to be generated by passing the other as an argument to a class template. Trying to break the circle in C++, we find that the mechanisms of forward declarations of a class template and an ordinary class are not sufficient. It is impossible to instantiate a template, if the argument passed to it is not a completely defined type.

To our awareness, the only true case of mutual collateral definitions in C++ is the definitions of members of a class or a struct. This is the reason why the concept of configuration in our approach, which does the simultaneous binding of roles to actors is therefore realized as a struct, as can be seen in Fig. 4.1.

Note how a **typedef** is used by the CONFIGURATION macro to store the name of the newly defined structure, and use it later in each of the ASSIGN_ROLE macros.

After macro expansion the definition of configuration `GraphConfig` in Fig. 2.3 becomes

```

struct GraphConfig {
    typedef GraphConfig self;
    typedef Vertex<self> V;
    typedef Edge<self> E;
    typedef Graph<self> G;
};

```

```

#define CONFIGURATION(name) struct name { \
    typedef name self;

#define END_CONFIGURATION };

#define ASSIGN_ROLE(role, actor) \
    typedef actor<self> role;

```

Figure 4.1: C macros to define a configuration

The structure `GraphConfig` defines three new types: `GraphConfig::V`, `GraphConfig::E` and `GraphConfig::C`. Since these types are members of the structure, the definitions are made in completely mutually-recursive manner. The recursive step in the type definitions is realized by passing `self` as a parameter to the class templates which generate the three new types. Finally, `self` is **typedefed** to `GraphConfig` which includes the newly defined types trio.

The particular solution in Fig. 4.1 relies also on a mutual recursive property between a structure and its members. We rely on the fact that the type `self`, **typedefed** to the type of the configuration class, can be used in the type definitions of its **typedef** members. Such a capability is not essential to our strategy. Instead of passing the single argument `self` as argument to actors, it is possible, but not as elegant, to pass the newly defined type members as such arguments.

Reexamining Fig. 4.1 we see that all actors are class templates, expecting a single **typename** argument. This argument is always a configuration, in which the roles are exported as **typedefs**. The macros in Fig. 4.2 are used to define base actors.

```

#define ACTOR \
    template <typename ConfigType> \
    class

#define INTERACTS_WITH(role) typedef typename ConfigType::role role

```

Figure 4.2: Macros for defining base actors

After macro expansion, actor `Edge` of Fig. 2.2 becomes:

```

template <typename ConfigType>
class Edge { public:
    typedef typename ConfigType::V V;
    Edge(V& v1, V& v2);           // a covariant constructor
    void set_first(V* v);         // a covariant method
    void set_second(V* v);       // another covariant method
    V& get_first();              // a method with a covariant return value
    // more definitions ...
    protected: V *first, *second; // two covariant fields
};

```

The class template `Edge` takes a configuration parameter named `ConfigType`. The configuration packages all the types (roles) with which the actor may interact. The `INTERACTS_WITH` directive simply expands the scope of a name of a role; a name which is initially a **typedef** in `ConfigType` is brought into the outer level; it can then be used directly, i.e., without the `ConfigType::` prefix, in the other data and function member definitions of the actor.

Fig. 4.3 presents the macros dedicated to the `SELF` role. As explained above in Sec. 4, the actor designates its preferred role using the directive `SELF_IS`, which is nothing but a **typedef** binding the name `Self` to the role `role`, passed as its argument. There is nothing in this binding to force the configuration to assign this role to requesting actor.

Using the `SELF_IS` directive makes the **typedef** `Self` available for e.g., binary method definitions, as can be seen by the macro expansion of (header of) actor `Point` of Fig. 3.1:

```
#define SELF_IS(role) typedef typename ConfigType::role Self
#define MY_ROLE(role) INTERACTS_WITH(role); SELF_IS(role)
```

Figure 4.3: Macros for Self role

```
template <typename ConfigType> \
    class Point { public:
        typedef typename ConfigType::P P;
        typedef typename ConfigType::P Self;
        // ...
};
```

Finally, Fig. 4.4 gives the macros for specializing an actor. The idea is that each instantiation of a class template representing a derived actor, is defined as inheriting from a base actor. The configuration (`ConfigType`) argument to the specializing actor is passed as an argument to the specialized base. The macro `SUPER_IS` relies on this argument passing to construct the name of the base class.

```
#define SPECIALIZED_ACTOR(name, base) \
    template <typename ConfigType> \
        class name: public base<ConfigType>

#define SUPER_IS(base) typedef base<ConfigType> Super
```

Figure 4.4: Macros for actor specialization

The following excerpt is from the macro expansion of actor `ColorPoint` ??

```
template <typename ConfigType> \
    class ColorPoint: public Point<ConfigType> { public:
        typedef base<ConfigType> Super;

        bool equal(const Self& other) const {
            return Super::equal(other) && color == other.color;
        }
};
```

The configuration `CPConfig` (Fig. 3.4) invokes the above template to create a class with three different names: `ColorPoint<CPConfig>`, `CPConfig::P` and `color_point`. This new class does not inherit from `Point<CPConfig>` (also known as `point`) but rather from class `Point<CPConfig>`.

The fact that `color_point` does not inherit from `point` is beneficial for run time performance. Suppose that a function member of actor `ColorPoint` calls a function member f introduced in the base actor `Point`. Then, we argue that there is no need to make f **virtual**, even if it is overridden in `ColorPoint`. The reason is that at class `color_point`, and more generally, in all valid instantiations of `ColorPoint`, the runtime type cannot be any instance of `Point`. In other words, `ColorPoint` sports a simple *implementation inheritance* (also called *extension inheritance*) from `Point`, with no inclusion polymorphism.

5 Discussion and Further Research

In a dynamically typed systems, covariance is not a problem. Our approach relies on this fact, on the observation that the computational process of instantiating a C++ template, is dynamically typed. A template definition does not include a “parameter type” or any other short signature that can be used to describe the kinds of type parameters it may receive. Instead, all type checks are done at “run time” when the template is instantiated. Since this “run time” occurs while the program compiles, any errors occurring then are still compile time errors.

We exploited this dual static-dynamic typing nature of templates, to effect a computational process, specifically the mutual substitution of arguments in several templates, which can create arbitrary hierarchies of covariant classes.

This computational process will fail every time the user tries to create an inconsistent (not type checkable in the type theoretical lingo) collection of class definitions. Luckily, all failures of this computational process would be reported as compile time errors, specifically ones related to a template parameter not matching the implicit assumptions of a template definition.

A different perspective on our solution is that the set $c(A)$ of assumptions, made by a collection of anchors (class templates) A , is in fact a type, which can be used for type-checking a collection of covariant classes. Even though we went around the difficult task of formalizing $c(A)$, we are still able to type check against it. In some sense, $c(A)$ encapsulates in it the structure of generalized matching. A major theoretical challenge is to find a formal foundation for the set of assumptions that a C++ template makes on its arguments. Such an achievement would be invaluable in the battle with the recalcitrant problem of templates in a separate compilation environment.

A difficulty that our approach, just as any other templates programming library, is that of code explosion—since most C++ compilers do not share code among different instantiations of templates. The problem of course is not as bad as with data structures libraries, and we are working on a programming methodology which reduces its impact.

Two principal lingual ingredients are required for applying our approach in other languages: mutually recursive definitions and templates. These two components exist in EIFFEL. In fact, the EIFFEL compiler applies a very sophisticated algorithm in its support for the language demand that *all* classes are mutually recursive. We have tried to implement our approach in EIFFEL, but so far failed. The difficulty was that since EIFFEL does not have type definitions in classes, we had to implement the mutual recursion process by passing multiple arguments to templates. It turns out that current EIFFEL compilers (including ISE's and SmallEiffel), do not support a mutual collateral and recursive relationship among template (generic classes in the EIFFEL lingo) arguments.⁵ The definitive language manual [18] is silent regarding this point.

Acknowledgments We thank Yuri Tsoglin for comments made on an earlier version of this manuscript. We pay tribute to Tatiana Surazhsky for her immense help in the implementation of the covariant library, and for many inspiring discussions.

References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.
- [2] K. Bruce. Increasing JAVA's expressiveness with thisType and match-bounded polymorphism. Available on the author's web page⁶, 1997.
- [3] K. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1996.
- [4] K. Bruce, A. Fiech, and L. Peteresen. Subtyping is not a good match for object-oriented languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming* [11].
- [5] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good “match” for object-oriented languages. In *Proceedings of the 11th European Conference on Object-Oriented Programming* [11].
- [6] K. B. Bruce, A. Fiech, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, number 952 in Lecture Notes in Computer Science, pages 27–51, Aarhus, Denmark, Aug. 7–11 1995. ECOOP'95, Springer Verlag.
- [7] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

⁵Our local version of the compilers crashed when faced with this challenge.

⁶www.cs.williams.edu/~kim

- [8] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Prog. Lang. Syst.*, 17(3):431–447, 1995.
- [9] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. *ACM Trans. Prog. Lang. Syst.*, 17(6):805–843, 1995.
- [10] D. Colnet and L. Liquori. *Match-O, a dialect of Eiffel with Match-Types*, pages 190–201. Prentice-Hall, Sydney, Australia, Nov. 20-23 2000.
- [11] ECOOP’97. *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.
- [13] J. Y. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 249–262, Santa Fe, New Mexico, Apr. 1998. USENIX.
- [14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.
- [15] C. Gotsman and V. Surazhsky. Guaranteed intersection-free polygon morphing. *Computers & Graphics*, 25(1):67–75, Feb. 2001.
- [16] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.
- [17] B. Meyer. *EIFFEL the Language*. Object-Oriented Series. Prentice-Hall, Hemel Hempstead, Hertfordshire, UK, 1992.
- [18] B. Meyer. *EIFFEL: The Language*. Object-Oriented Series. Prentice-Hall, 1992.
- [19] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, number 1628 in Lecture Notes in Computer Science, pages 279–303, Lisbon, Portugal, June 14–18 1999. ECOOP’99, Springer Verlag.
- [20] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standart Template Library*. Addison-Wesley, 1996.
- [21] S. M. Omohundro. *The Sather 1.0 Specification*, Jan. 1994.
- [22] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.
- [23] R. Rinat. Type-safe covariant specialization with generalized matching. In *The 7th International Workshop on Foundations of Object-Oriented Languages, FOOL 7*, Jan. 2000.
- [24] G. Steele. *Common Lisp the language*. Digital, 1990.
- [25] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991.
- [26] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.
- [27] C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. Technical Report TR-93-064, The International Computer Science Institute, Berkeley, Ca, Nov. 1993.
- [28] D. A. Watt. *Programming Language: Concepts and Paradigms*. Prentice-Hall, 1990.
- [29] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.